De Lijn Realtime

By Freek De Sagher S0171876
project for Distributed Systems 2019-2020

Contents

Installation and running

Automatic installation

This section explains how to install all the dependencies for both parts of the project using one script. If somehow this script fails, see the manual installation section for a detailed walktrough of a manual installation.

The project can be installed using the install.sh script. This script requires sudo privileges to work.

```
sudo ./install.sh
```

After installing, go to the Running the project section for the information on how to start the back- and front-end server.

Manual installation

Check this section if there were problems with the install script.

Back-end

The back-end of this application is built with Python and the Flask framework. To install the project, navigate to the WebServices folder. To be able to install the back-end, Python 3 and pip are required. Install the requirements by running

> pip install -r requirements.txt

When this is done, the back-end server can be started with

> python app.py or python3 app.py

The webserver will start on 127.0.0.1:5000 by default. Going to this address in the browser will show a brief documentation of the API. This documentation is also given in this document.

Front-end

The front-end of this application is built with the React framework. To install this part of the project, go to web-services-front. To be able to install the front-end npm is required. Once npm is installed, run

> npm install

To start the development server run

> npm run start

By default, the front end server will start on 127.0.0.1:3000

Running the project

Running the project is quite simple. The script start_frontend.sh starts the front-end server and the start_backend.sh file starts the back-end server. No sudo rights are required to run these scripts. If you want to start the servers manually, you can look into the scripts. They both contain one line that also can be executed in the terminal without the use of the scripts.

Configuration

The back- and front-end both have a configuration file to set some global variables.

Back-end

The back-end configuration file can be found in the WebServices directory and is called config.py. This configuration file has three entries:

> DE_LIJN_API_KEY: the API key used for requests to the service of De Lijn

> WEATHER_API_KEY: the API key used for requests to the weather service

> TOMTOM_API_KEY: the API key used for requests to the TomTom Routing API.

Front-end

The configuration file config.jsx can be found in the source folder in the webservices-front directory. This configuration file has three entries:

> API_BASE: the base url of the API. The default example is 127.0.0.1:5000. Change this if the address of the API changes.

> INTERVAL: the time between an API call that updates the vehicle locations. The time is in milliseconds. Default value is 10000.

> TOMTOM_API_KEY: the API key for the TomTom routing API.

API Structure

In this section, all the possible API can be found.

Error structure

Sometimes exceptions can occur (De Lijn API doesn't always return things, max calls exceeded...). An http error will be thrown with the following structure

```
{
    "message": "message" ('Not Found' for example),
    "error": "additional error message",
    "status": 404 (for example)
}
```

GET /entities

Get all the entities of De Lijn. Each returned entity has a number (the identifier of the entity) and a description. If the function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/entities
[
    {
        "number": 1,
        "desc": "Antwerpen"
    },
    ...
]
```

GET /lines

Get all lines of De Lijn. Each line has a number, an entity number (for example 1 = Antwerp), a description, a type ("tram" or "bus") and a service type. If the function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/lines
[
    {
        "number": 2,
        "entityNumber": 1,
        "desc": "Hoboken - P+R Merksem",
        "type": "tram",
        "serviceType": "normaal"
    },
    ...
]
```

GET /villages

Get all villages of De Lijn. Each village has an id and a name. If this function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/villages
[
    ...
    {
        "id": 553,
        "name": "Sint-Katelijne-Waver"
    },
    {
        "id": 554,
        "name": "Oppuurs"
    },
    {
        "id": 555,
        "name": "Sint-Amands"
    },
    ...
]
```

GET /stops

Get all tram and bus stops of De Lijn. This function takes a lot of time (~40 seconds). If the function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/stops
[
    {
        "number": "101000",
        "villageName": "Wilrijk",
        "desc": "A. Chantrainestraat",
        "latlng": [
        51.16388937021345,
        4.392073389160737
```

```
    ]
    },
    ...
]
```

GET /lines/{entity_number}/{line_number}/{direction}

This function returns all stops of a certain line.
Parameters

- entity_number: positive integer; represents the entities of De Lijn. The accepted numbers are [1, 5]
- line_number: positive integer; represents the line number, ex. 32
- direction: string; can only be 'HEEN' or 'TERUG'

If this function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/lines/1/253/HEEN
[
    ...
    {
        "number": 107809,
        "desc": "Kerkhof",
        "village": "Oppuurs",
        "latlng": [
            51.07022693785234,
            4.252991607174736
        ]
    },
    {
        "number": 104449,
        "desc": "Zeutestraat",
        "village": "Puurs",
        "latlng": [
            51.07329594009163,
            4.266924020775759
        ]
    },
    ...
]
```

GET /weather/{latitude}/{longitude}

Get the weather of a certain geolocation.
Parameters

- latitude: positive float; the latitude of the geolocation.
- longitude: positive float; the longitude of the geolocation.

Return types

- clouds: percentage
- humidity: percentage
- windspeed: float in m/s
- temp: float in Celsius

If this function succeeds, the response code will be 200.

```
Example URL: 127.0.0.1:5000/weather/51.4/4.4
{
    "clouds": 88,
    "humidity": 58,
    "windspeed": 8.92,
    "desc": "overcast clouds",
    "temp": 18.76
}
```

GET /vehicles/{entity_number}/{line_number}/{direction}

Get the geo locations of the vehicles of a certain line.
Parameters

- entity_number: positive integer
- line_number: positive integer
- direction: string; can only be 'HEEN' or 'TERUG'

Return types

- First list item: latitude
- Second list item: longitude

```
Example URL: 127.0.0.1:5000/vehicles/1/32/HEEN
[
  [51.15562, 4.43622],
  [51.19465, 4.42364],
  [51.21025, 4.41818]
]
```

Design choices

Back-end

The back-end was written in Python using the Flask framework. This was mandatory in the assignment so this was an obvious choice. The back-end server makes use of three external API's: one

for the data of De Lijn, one for the weather, and one for routing. The API of De Lijn was obligated. For the weather API, I chose OpenWeatherMap. This webservices uses open data, a concept I really like and want to use. Furthermore, this service offered the most free API calls per day and the service is easy to use.
For the routing part of this application, I chose for the TomTom Routing API. First, I tried to use OSRM. This was however not the best option. The free service of OSRM is based on their demo server. This server was not reliable at all and sometimes didn't even return data. The only way to use this service, was buying or creating your own server. That's why I started looking for better alternatives. I found out TomTom provided the best service for routing with a fairly large amount of API calls per day.

Front-end

The front-end is completely decoupled from the back-end. To demonstrate this, I didn't use Flask again for the front-end part, but I chose for a completely different framework. I picked the ReactJS framework because I already gained some experience with this last summer. I find it much cleaner to use a component based system instead of writing massive HTML and javascript files. Each component kind of stands on its own and this gives in my opinion a way better overview over the code.
The requests in the front-end are done by a library called Axios. This is a promise based HTTP client. I picked this library because, once again, I gained some experience with it last summer. I think Axios makes creating requests for webservers quite easy.
The map is created using LeafletJS. This library makes it very easy to add an OpenStreetMap based map to your webapplication. The documentation of this library is very useful and the library is straightforward to use. Things as markers and popups require not that much work. I also used the Leaflet-Routing-Machine extension. This Leaflet extension makes it easier to draw routes on your map.
Lastly, I used Bootstrap 4 to help with the CSS of this application. We used Bootstrap in a project last year, so this accelerated the development of the webservice.
The weather only gets fetched from the API when a stop marker is clicked. The OpenWeatherMap API only allows 60 calls per minute. Using this method, the application does not generate API requests when no marker is clicked.

Used tools

Back-end

- De Lijn API for the bus and tram data
- TomTom Routing API for routing
- OpenWeatherMap for the weather data
- Flask Restful to create the rest API.
- Flask Cors to configure CORS on the API

Front-end

- Leaflet for map creation
- Leaflet Routing Machine for route calculation and display
- LRM TomTom package to use TomTom api with leaflet routing machine
- Axios for http requests
- React core framework used for the front end
- OpenStreetMap data data used to display the Leaflet map
- Bootstrap 4 for easier CSS
- JQuery needed for Bootstrap